```
------------------------------------------------------------------------------
--                                                                          --
--                         GNAT COMPILER COMPONENTS                         --
--                                                                          --
--                                  P A R                                   --
--                                                                          --
--                                  B o d y                                 --
--                                                                          --
--                              $Revision: 1.116 $                          --
--                                                                          --
--          Copyright (C) 1992-1998 Free Software Foundation, Inc.          --
--                                                                          --
-- GNAT is free software;  you can  redistribute it  and/or modify it under --
-- terms of the  GNU General Public License as published  by the Free Soft- --
-- ware  Foundation;  either version 2,  or (at your option) any later ver- --
-- sion.  GNAT is distributed in the hope that it will be useful, but WITH- --
-- OUT ANY WARRANTY;  without even the  implied warranty of MERCHANTABILITY --
-- or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License --
-- for  more details.  You should have  received  a copy of the GNU General --
-- Public License  distributed with GNAT;  see file COPYING.  If not, write --
-- to  the Free Software Foundation,  59 Temple Place - Suite 330,  Boston, --
-- MA 02111-1307, USA.                                                      --
--                                                                          --
-- GNAT was originally developed  by the GNAT team at  New York University. --
-- It is now maintained by Ada Core Technologies Inc (http://www.gnat.com). --
--                                                                          --
------------------------------------------------------------------------------

with Atree;     use Atree;
with Casing;    use Casing;
with Csets;     use Csets;
with Debug;     use Debug;
with Elists;    use Elists;
with Errout;    use Errout;
with Fname;     use Fname;
with Lib;       use Lib;
with Namet;     use Namet;
with Nlists;    use Nlists;
with Nmake;     use Nmake;
with Opt;       use Opt;
with Output;    use Output;
with Scans;     use Scans;
with Scn;       use Scn;
with Sinput;    use Sinput;
with Sinput.L;  use Sinput.L;
with Sinfo;     use Sinfo;
with Snames;    use Snames;
with Style;
with Table;

function Par (Configuration_Pragmas : Boolean) return List_Id is

   Num_Library_Units : Natural := 0;
   --  Count number of units parsed (relevant only in syntax check only mode,
   --  since in semantics check mode only a single unit is permitted anyway)

   Unit_Node : Node_Id;
   --  Stores compilation unit node for current unit

   Save_Ada_83_Mode : Boolean;
   --  Saves state of Ada 83 mode switch for restore on exit (since it may
   --  get reset by occurrence of the Ada_83 or Ada_95 pragmas).
```

```
   Loop_Block_Count : Nat := 0;
   --  Counter used for constructing loop/block names (see the routine
   --  Par.Ch5.Get_Loop_Block_Name)

   ----------------------
   -- Error Recovery --
   ----------------------

   --  When an error is encountered, a call is made to one of the Error_Msg
   --  routines to record the error. If the syntax scan is not derailed by the
   --  error (e.g. a complaint that logical operators are inconsistent in an
   --  EXPRESSION), then control returns from the Error_Msg call, and the
   --  parse continues unimpeded.

   --  If on the other hand, the Error_Msg represents a situation from which
   --  the parser cannot recover locally, the exception Error_Resync is raised
   --  immediately after the call to Error_Msg. Handlers for Error_Resync
   --  are located at strategic points to resynchronize the parse. For example,
   --  when an error occurs in a statement, the handler skips to the next
   --  semicolon and continues the scan from there.

   --  Each parsing procedure contains a note with the heading "Error recovery"
   --  which shows if it can propagate the Error_Resync exception. In order
   --  not to propagate the exception, a procedure must either contain its own
   --  handler for this exception, or it must not call any other routines which
   --  propagate the exception.

   --  Note: the arrangement of Error_Resync handlers is such that it should
   --  never be possible to transfer control through a procedure which made
   --  an entry in the scope stack, invalidating the contents of the stack.

   Error_Resync : exception;
   --  Exception raised on error that is not handled locally, see above.

   Last_Resync_Point : Source_Ptr;
   --  The resynchronization routines in Par.Sync run a risk of getting
   --  stuck in an infinite loop if they do not skip a token, and the caller
   --  keeps repeating the same resync call. On the other hand, if they skip
   --  a token unconditionally, some recovery opportunities are missed. The
   --  variable Last_Resync_Point records the token location previously set
   --  by a Resync call, and if a subsequent Resync call occurs at the same
   --  location, then the Resync routine does guarantee to skip a token.

   -------------------------------------------
   -- Handling Semicolon Used in Place of IS --
   -------------------------------------------

   --  The following global variables are used in handling the error situation
   --  of using a semicolon in place of IS in a subprogram declaration as in:

   --     procedure X (Y : Integer);
   --        Q : Integer;
   --     begin
   --        ...
   --     end;

   --  The two contexts in which this can appear are at the outer level, and
   --  within a declarative region. At the outer level, we know something is
   --  wrong as soon as we see the Q (or begin, if there are no declarations),
   --  and we can immediately decide that the semicolon should have been IS.

   --  The situation in a declarative region is more complex. The declaration
   --  of Q could belong to the outer region, and we do not know that we have
```

```
   --  an error until we hit the begin. It is still not clear at this point
   --  from a syntactic point of view that something is wrong, because the
   --  begin could belong to the enclosing subprogram or package. However, we
   --  can incorporate a bit of semantic knowledge and note that the body of
   --  X is missing, so we definitely DO have an error. We diagnose this error
   --  as semicolon in place of IS on the subprogram line.

   --  There are two styles for this diagnostic. If the begin immediately
   --  follows the semicolon, then we can place a flag (IS expected) right
   --  on the semicolon. Otherwise we do not detect the error until we hit
   --  the begin which refers back to the line with the semicolon.

   --  To control the process in the second case, the following global
   --  variables are set to indicate that we have a subprogram declaration
   --  whose body is required and has not yet been found. The prefix SIS
   --  stands for "Subprogram IS" handling.

   SIS_Entry_Active : Boolean;
   --  Set True to indicate that an entry is active (i.e. that a subprogram
   --  declaration has been encountered, and no body for this subprogram has
   --  been encountered). The remaining fields are valid only if this is True.

   SIS_Labl : Node_Id;
   --  Subprogram designator

   SIS_Sloc : Source_Ptr;
   --  Source location of FUNCTION/PROCEDURE keyword

   SIS_Ecol : Column_Number;
   --  Column number of FUNCTION/PROCEDURE keyword

   SIS_Semicolon_Sloc : Source_Ptr;
   --  Source location of semicolon at end of subprogram declaration

   SIS_Declaration_Node : Node_Id;
   --  Pointer to tree node for subprogram declaration

   SIS_Missing_Semicolon_Message : Error_Msg_Id;
   --  Used to save message ID of missing semicolon message (which will be
   --  modified to missing IS if necessary). Set to No_Error_Msg in the
   --  normal (non-error) case.

   --  Five things can happen to an active SIS entry

   --   1. If a BEGIN is encountered with an SIS entry active, then we have
   --   exactly the situation in which we know the body of the subprogram is
   --   missing. After posting an error message, we change the spec to a body,
   --   rechaining the declarations that intervened between the spec and BEGIN.

   --   2. Another subprogram declaration or body is encountered. In this
   --   case the entry gets overwritten with the information for the new
   --   subprogram declaration. We don't catch some nested cases this way,
   --   but it doesn't seem worth the effort.

   --   3. A nested declarative region (e.g. package declaration or package
   --   body) is encountered. The SIS active indication is reset at the start
   --   of such a nested region. Again, like case 2, this causes us to miss
   --   some nested cases, but it doesn't seen worth the effort to stack and
   --   unstack the SIS information. Maybe we will reconsider this if we ever
   --   get a complaint about a missed case :-)

   --   4. We encounter a valid pragma INTERFACE or IMPORT that effectively
   --   supplies the missing body. In this case we reset the entry.
```

```
--    5. We encounter the end of the declarative region without encoutering
--    a BEGIN first. In this situation we simply reset the entry. We know
--    that there is a missing body, but it seems more reasonable to let the
--    later semantic checking discover this.

-------------------------------------------
-- Handling IS Used in Place of Semicolon --
-------------------------------------------

--  This is a somewhat trickier situation, and we can't catch it in all
--  cases, but we do our best to detect common situations resulting from
--  a "cut and paste" operation which forgets to change the IS to semicolon.
--  Consider the following example:

--    package body X is
--      procedure A;
--      procedure B is
--      procedure C;
--      ...
--      procedure D is
--      begin
--         ...
--      end;
--    begin
--      ...
--    end;

--  The trouble is that the section of text from PROCEDURE B through END;
--  consitutes a valid procedure body, and the danger is that we find out
--  far too late that something is wrong (indeed most compilers will behave
--  uncomfortably on the above example).

--  We have two approaches to helping to control this situation. First we
--  make every attempt to avoid swallowing the last END; if we can be
--  sure that some error will result from doing so. In particular, we won't
--  accept the END; unless it is exactly correct (in particular it must not
--  have incorrect name tokens), and we won't accept it if it is immediately
--  followed by end of file, WITH or SEPARATE (all tokens that unmistakeably
--  signal the start of a compilation unit, and which therefore allow us to
--  reserve the END; for the outer level.) For more details on this aspect
--  of the handling, see package Par.Endh.

--  If we can avoid eating up the END; then the result in the absense of
--  any additional steps would be to post a missing END referring back to
--  the subprogram with the bogus IS. Similarly, if the enclosing package
--  has no BEGIN, then the result is a missing BEGIN message, which again
--  refers back to the subprogram header.

--  Such an error message is not too bad (it's already a big improvement
--  over what many parsers do), but it's not ideal, because the declarations
--  following the IS have been absorbed into the wrong scope. In the above
--  case, this could result for example in a bogus complaint that the body
--  of D was missing from the package.

--  To catch at least some of these cases, we take the following additional
--  steps. First, a subprogram body is marked as having a suspicious IS if
--  the declaration line is followed by a line which starts with a symbol
--  that can start a declaration in the same column, or to the left of the
--  column in which the FUNCTION or PROCEDURE starts (normal style is to
--  indent any declarations which really belong a subprogram). If such a
--  subprogram encounters a missing BEGIN or missing END, then we decide
--  that the IS should have been a semicolon, and the subprogram body node
```

```
   --  is marked (by setting the Bad_Is_Detected flag true. Note that we do
   --  not do this for library level procedures, only for nested procedures,
   --  since for library level procedures, we must have a body.

   --  The processing for a declarative part checks to see if the last
   --  declaration scanned is marked in this way, and if it is, the tree
   --  is modified to reflect the IS being interpreted as a semicolon.

   ------------------------------------------------
   -- Parser Type Definitions and Control Variables --
   ------------------------------------------------

   --  The following variable and associated type declaration are used by the
   --  expression parsing routines to return more detailed information about
   --  the categorization of a parsed expression.

   type Expr_Form_Type is (
      EF_Simple_Name,  -- Simple name, i.e. possibly qualified identifier
      EF_Name,         -- Simple expression which could also be a name
      EF_Simple,       -- Simple expression which is not call or name
      EF_Range_Attr,   -- Range attribute reference
      EF_Non_Simple);  -- Expression that is not a simple expression

   Expr_Form : Expr_Form_Type;

   --  The following type is used for calls to P_Subprogram, P_Package, P_Task,
   --  P_Protected to indicate which of several possibilities is acceptable.

   type Pf_Rec is record
      Spcn : Boolean;                  -- True if specification OK
      Decl : Boolean;                  -- True if declaration OK
      Gins : Boolean;                  -- True if generic instantiation OK
      Pbod : Boolean;                  -- True if proper body OK
      Rnam : Boolean;                  -- True if renaming declaration OK
      Stub : Boolean;                  -- True if body stub OK
      Fil1 : Boolean;                  -- Filler to fill to 8 bits
      Fil2 : Boolean;                  -- Filler to fill to 8 bits
   end record;
   pragma Pack (Pf_Rec);

   function T return Boolean renames True;
   function F return Boolean renames False;

   Pf_Decl_Gins_Pbod_Rnam_Stub : constant Pf_Rec :=
                                    Pf_Rec'(F, T, T, T, T, T, F, F);
   Pf_Decl                     : constant Pf_Rec :=
                                    Pf_Rec'(F, T, F, F, F, F, F, F);
   Pf_Decl_Gins_Pbod_Rnam      : constant Pf_Rec :=
                                    Pf_Rec'(F, T, T, T, T, F, F, F);
   Pf_Decl_Pbod                : constant Pf_Rec :=
                                    Pf_Rec'(F, T, F, T, F, F, F, F);
   Pf_Pbod                     : constant Pf_Rec :=
                                    Pf_Rec'(F, F, F, T, F, F, F, F);
   Pf_Spcn                     : constant Pf_Rec :=
                                    Pf_Rec'(T, F, F, F, F, F, F, F);
   --  The above are the only allowed values of Pf_Rec arguments

   type SS_Rec is record
      Eftm : Boolean;      -- ELSIF can terminate sequence
      Eltm : Boolean;      -- ELSE can terminate sequence
      Extm : Boolean;      -- EXCEPTION can terminate sequence
      Ortm : Boolean;      -- OR can terminate sequence
      Sreq : Boolean;      -- at least one statement required
```

```
   Tatm : Boolean;      -- THEN ABORT can terminate sequence
   Whtm : Boolean;      -- WHEN can terminate sequence
   Unco : Boolean;      -- Unconditional terminate after one statement
end record;
pragma Pack (SS_Rec);

SS_Eftm_Eltm_Sreq : constant SS_Rec := SS_Rec'(T, T, F, F, T, F, F, F);
SS_Eltm_Ortm_Tatm : constant SS_Rec := SS_Rec'(F, T, F, T, F, T, F, F);
SS_Extm_Sreq      : constant SS_Rec := SS_Rec'(F, F, T, F, T, F, F, F);
SS_None           : constant SS_Rec := SS_Rec'(F, F, F, F, F, F, F, F);
SS_Ortm_Sreq      : constant SS_Rec := SS_Rec'(F, F, F, T, T, F, F, F);
SS_Sreq           : constant SS_Rec := SS_Rec'(F, F, F, F, T, F, F, F);
SS_Sreq_Whtm      : constant SS_Rec := SS_Rec'(F, F, F, F, T, F, T, F);
SS_Whtm           : constant SS_Rec := SS_Rec'(F, F, F, F, F, F, T, F);
SS_Unco           : constant SS_Rec := SS_Rec'(F, F, F, F, F, F, F, T);

type End_Action_Type is (
--  Type used to describe the result of the Pop_End_Context call

   Accept_As_Scanned,
   --  Current end sequence is entirely c correct. In this case Token and
   --  the scan pointer are left pointing past the end sequence (i.e. they
   --  are unchanged from the values set on entry to Pop_End_Context).

   Insert_And_Accept,
   --  Current end sequence is to be left in place to satisfy some outer
   --  scope. Token and the scan pointer are set to point to the end
   --  token, and should be left there. A message has been generated
   --  indicating a missing end sequence. This status is also used for
   --  the case when no end token is present.

   Skip_And_Accept,
   --  The end sequence is incorrect (and an error message has been
   --  posted), but it will still be accepted. In this case Token and
   --  the scan pointer point back to the end token, and the caller
   --  should skip past the end sequence before proceeding.

   Skip_And_Reject);
   --  The end sequence is judged to belong to an unrecognized inner
   --  scope. An appropriate message has been issued and the caller
   --  should skip past the end sequence and then proceed as though
   --  no end sequence had been encountered.

End_Action : End_Action_Type;
--  The variable set by Pop_End_Context call showing which of the four
--  decisions described above is judged the best.

Label_List : Elist_Id;
--  List of label nodes for labels appearing in the current compilation.
--  Used by Par.Labl to construct the corresponding implicit declarations.

-----------------
-- Scope Table --
-----------------

--  The scope table, also referred to as the scope stack, is used to
--  record the current scope context. It is organized as a stack, with
--  inner nested entries corresponding to higher entries on the stack.
--  An entry is made when the parser encounters the opening of a nested
--  construct (such as a record, task, package etc.), and then package
--  Par.Endh uses this stack to deal with END lines (including properly
--  dealing with END nesting errors).
```

```
type SS_End_Type is
--   Type of end entry required for this scope. The last two entries are
--   used only in the subprogram body case to mark the case of a suspicious
--   IS, or a bad IS (i.e. suspicions confirmed by missing BEGIN or END).
--   See separate section on dealing with IS used in place of semicolon.
--   Note that for many purposes E_Name, E_Suspicious_Is and E_Bad_Is are
--   treated the same (E_Suspicious_Is and E_Bad_Is are simply special cases
--   of E_Name). They are placed at the end of the enumeration so that a
--   test for >= E_Name catches all three cases efficiently.

   (E_Dummy,            -- dummy entry at outer level
    E_Case,             -- END CASE;
    E_If,               -- END IF;
    E_Loop,             -- END LOOP;
    E_Record,           -- END RECORD;
    E_Select,           -- END SELECT;
    E_Name,             -- END [name];
    E_Suspicious_Is,    -- END [name]; (case of suspicious IS)
    E_Bad_Is);          -- END [name]; (case of bad IS)

--   The following describes a single entry in the scope table

type Scope_Table_Entry is record
   Etyp : SS_End_Type;
   --  Type of end entry, as per above description

   Lreq : Boolean;
   --  A flag indicating whether the label, if present, is required to
   --  appear on the end line. It is referenced only in the case of
   --  Etyp = E_Name or E_Suspicious_Is where the name may or may not be
   --  required (yes for labeled block, no in other cases). Note that for
   --  all cases except begin, the question of whether a label is required
   --  can be determined from the other fields (for loop, it is required if
   --  it is present, and for the other constructs it is never required or
   --  allowed).

   Ecol : Column_Number;
   --  Contains the absolute column number (with tabs expanded) of the
   --  the expected column of the end assuming normal Ada indentation
   --  usage. If the RM_Column_Check mode is set, this value is used for
   --  generating error messages about indentation. Otherwise it is used
   --  only to control heuristic error recovery actions.

   Labl : Node_Id;
   --  This field is used only for the LOOP and BEGIN cases, and is the
   --  Node_Id value of the label name. For all cases except child units,
   --  this value is an entity whose Chars field contains the name pointer
   --  that identifies the label uniquely. For the child unit case the Labl
   --  field references an N_Defining_Program_Unit_Name node for the name.
   --  For cases other than LOOP or BEGIN, the Label field is set to Error,
   --  indicating that it is an error to have a label on the end line.

   Decl : List_Id;
   --  Points to the list of declarations (i.e. the declarative part)
   --  associated with this construct. It is set only in the END [name]
   --  cases, and is set to No_List for all other cases which do not have a
   --  declarative unit associated with them. This is used for determining
   --  the proper location for implicit label declarations.

   Sloc : Source_Ptr;
   --  Source location of the opening token of the construct. This is
   --  used to refer back to this line in error messages (such as missing
   --  or incorrect end lines). The Sloc field is not used, and is not set,
```

```
      --  if a label is present (the Labl field provides the text name of the
      --  label in this case, which is fine for error messages).

      S_Is : Source_Ptr;
      --  S_Is is relevant only if Etyp is set to E_Suspicious_Is or
      --  E_Bad_Is. It records the location of the IS that is considered
      --  to be suspicious.

      Junk : Boolean;
      --  A boolean flag that is set true if the opening entry is the dubious
      --  result of some prior error, e.g. a record entry where the record
      --  keyword was missing. It is used to suppress the issuing of a
      --  corresponding junk complaint about the end line (we do not want
      --  to complain about a missing end record when there was no record).
   end record;

   --  The following declares the scope table itself. The Last field is the
   --  stack pointer, so that Scope.Table (Scope.Last) is the top entry. The
   --  oldest entry, at Scope_Stack (0), is a dummy entry with Etyp set to
   --  E_Dummy, and the other fields undefined. This dummy entry ensures that
   --  Scope_Stack (Scope_Stack_Ptr).Etyp can always be tested, and that the
   --  scope stack pointer is always in range.

   package Scope is new Table.Table (
     Table_Component_Type => Scope_Table_Entry,
     Table_Index_Type     => Int,
     Table_Low_Bound      => 0,
     Table_Initial        => 50,
     Table_Increment      => 100,
     Table_Name           => "Scope");

   ----------------------------------
   -- Parsing Routines by Chapter --
   ----------------------------------

   --  Uncommented declarations in this section simply parse the construct
   --  corresponding to their name, and return an ID value for the Node or
   --  List that is created.

   package Ch2 is
      function P_Identifier                                  return Node_Id;
      function P_Pragma                                      return Node_Id;

      function P_Pragmas_Opt return List_Id;
      --  This function scans for a sequence of pragmas in other than a
      --  declaration sequence or statement sequence context. All pragmas
      --  can appear except pragmas Assert and Debug, which are only allowed
      --  in a declaration or statement sequence context.

      procedure P_Pragmas_Misplaced;
      --  Skips misplaced pragmas with a complaint

      procedure P_Pragmas_Opt (List : List_Id);
      --  Parses optional pragmas and appends them to the List
   end Ch2;

   package Ch3 is
      Missing_Begin_Msg : Error_Msg_Id;
      --  This variable is set by a call to P_Declarative_Part. Normaly it
      --  is set to No_Error_Msg, indicating that no special processing is
      --  required by the caller. The special case arises when a statement
      --  is found in the sequence of declarations. In this case the Id of
      --  the message issued ("declaration expected") is preserved in this
```

```
      --  variable, then the caller can change it to an appropriate missing
      --  begin message if indeed the BEGIN is missing.

      function P_Access_Definition                    return Node_Id;
      function P_Access_Type_Definition               return Node_Id;
      function P_Array_Type_Definition                return Node_Id;
      function P_Basic_Declarative_Items              return List_Id;
      function P_Constraint_Opt                       return Node_Id;
      function P_Declarative_Part                     return List_Id;
      function P_Defining_Identifier                  return Node_Id;
      function P_Discrete_Choice_List                 return List_Id;
      function P_Discrete_Range                       return Node_Id;
      function P_Discrete_Subtype_Definition          return Node_Id;
      function P_Known_Discriminant_Part_Opt          return List_Id;
      function P_Signed_Integer_Type_Definition       return Node_Id;
      function P_Range                                return Node_Id;
      function P_Range_Or_Subtype_Mark                return Node_Id;
      function P_Range_Constraint                     return Node_Id;
      function P_Record_Definition                    return Node_Id;
      function P_Subtype_Indication                   return Node_Id;
      function P_Subtype_Mark                         return Node_Id;
      function P_Subtype_Mark_Resync                  return Node_Id;
      function P_Unknown_Discriminant_Part_Opt        return Boolean;

      procedure P_Component_Items (Decls : List_Id);
      --  Scan out one or more component items and append them to the
      --  given list. Only scans out more than one declaration in the
      --  case where the source has a single declaration with multiple
      --  defining identifiers.

      function Init_Expr_Opt (P : Boolean := False) return Node_Id;
      --  If an initialization expression is present (:= expression), then
      --  it is scanned out and returned, otherwise Empty is returned if no
      --  initialization expression is present. This procedure also handles
      --  certain common error cases cleanly. The parameter P indicates if
      --  a right paren can follow the expression (default = no right paren
      --  allowed).

      procedure Skip_Declaration (S : List_Id);
      --  Used when scanning statements to skip past a mispaced declaration
      --  The declaration is scanned out and appended to the given list.
      --  Token is known to be a declaration token (in Token_Class_Declk)
      --  on entry, so there definition is a declaration to be scanned.

      function P_Subtype_Indication (Subtype_Mark : Node_Id) return Node_Id;
      --  This version of P_Subtype_Indication is called when the caller has
      --  already scanned out the subtype mark which is passed as a parameter.

      function P_Subtype_Mark_Attribute (Type_Node : Node_Id) return Node_Id;
      --  Parse a subtype mark attribute. The caller has already parsed the
      --  subtype mark, which is passed in as the argument, and has checked
      --  that the current token is apostrophe.

   end Ch3;

   package Ch4 is
      function P_Aggregate                            return Node_Id;
      function P_Expression                           return Node_Id;
      function P_Expression_No_Right_Paren            return Node_Id;
      function P_Expression_Or_Range_Attribute        return Node_Id;
      function P_Function_Name                        return Node_Id;
      function P_Name                                 return Node_Id;
      function P_Qualified_Simple_Name                return Node_Id;
```

```
   function P_Qualified_Simple_Name_Resync          return Node_Id;
   function P_Simple_Expression                     return Node_Id;
   function P_Simple_Expression_Or_Range_Attribute return Node_Id;

   function P_Qualified_Expression
     (Subtype_Mark : Node_Id)
      return         Node_Id;
   --  This routine scans out a qualified expression when the caller has
   --  already scanned out the name and apostrophe of the construct.

end Ch4;

package Ch5 is

   function P_Statement_Name (Name_Node : Node_Id) return Node_Id;
   --  Given a node representing a name (which is a call), converts it
   --  to the syntactically corresponding procedure call statement.

   function P_Sequence_Of_Statements (SS_Flags : SS_Rec) return List_Id;
   --  The argument indicates the acceptable termination tokens.
   --  See body in Par.Ch5 for details of the use of this parameter.

   procedure Parse_Decls_Begin_End (Parent : Node_Id);
   --  Parses declarations and handled statement sequence, setting
   --  fields of Parent node appropriately.

end Ch5;

package Ch6 is
   function P_Designator                            return Node_Id;
   function P_Defining_Program_Unit_Name            return Node_Id;
   function P_Formal_Part                           return List_Id;
   function P_Parameter_Profile                     return List_Id;
   function P_Return_Statement                      return Node_Id;
   function P_Subprogram_Specification              return Node_Id;

   procedure P_Mode (Node : Node_Id);
   --  Sets In_Present and/or Out_Present flags in Node scanning past
   --  IN, OUT or IN OUT tokens in the source.

   function P_Subprogram (Pf_Flags : Pf_Rec)        return Node_Id;
   --  Scans out any construct starting with either of the keywords
   --  PROCEDURE or FUNCTION. The parameter indicates which possible
   --  possible kinds of construct (body, spec, instantiation etc.)
   --  are permissible in the current context.

end Ch6;

package Ch7 is
   function P_Package (Pf_Flags : Pf_Rec) return Node_Id;
   --  Scans out any construct starting with the keyword PACKAGE. The
   --  parameter indicates which possible kinds of construct (body, spec,
   --  instantiation etc.) are permissible in the current context.
end Ch7;

package Ch8 is
   function P_Use_Clause                            return Node_Id;
end Ch8;

package Ch9 is
   function P_Abort_Statement                       return Node_Id;
   function P_Abortable_Part                        return Node_Id;
   function P_Accept_Statement                      return Node_Id;
```

```
   function P_Delay_Statement                          return Node_Id;
   function P_Entry_Body                               return Node_Id;
   function P_Protected                                return Node_Id;
   function P_Requeue_Statement                        return Node_Id;
   function P_Select_Statement                         return Node_Id;
   function P_Task                                     return Node_Id;
   function P_Terminate_Alternative                    return Node_Id;
end Ch9;

package Ch10 is
   function P_Compilation_Unit                         return Node_Id;
end Ch10;

package Ch11 is
   function P_Handled_Sequence_Of_Statements           return Node_Id;
   function P_Raise_Statement                          return Node_Id;

   function Parse_Exception_Handlers                   return List_Id;
   --  Parses the partial construct EXCEPTION followed by a list of
   --  exception handlers which appears in a number of productions,
   --  and returns the list of exception handlers.

end Ch11;

package Ch12 is
   function P_Generic                                  return Node_Id;
   function P_Generic_Actual_Part_Opt                  return List_Id;
end Ch12;

package Ch13 is
   function P_Representation_Clause                    return Node_Id;

   function P_Code_Statement (Subtype_Mark : Node_Id) return Node_Id;
   --  Function to parse a code statement. The caller has scanned out
   --  the name to be used as the subtype mark (but has not checked that
   --  it is suitable for use as a subtype mark, i.e. is either an
   --  identifier or a selected component). The current token is an
   --  apostrophe and the following token is either a left paren or
   --  RANGE (the latter being an error to be caught by P_Code_Statement.
end Ch13;

--  Note: the parsing for annexe J features (i.e. obsolescent features)
--  is found in the logical section where these features would be if
--  they were not obsolescent. In particular:

--    Delta constraint is parsed by P_Delta_Constraint (3.5.9)
--    At clause is parsed by P_At_Clause (13.1)
--    Mod clause is parsed by P_Mod_Clause (13.5.1)

-----------------------
-- End Handling --
-----------------------

--  Routines for handling end lines, including scope recovery

package Endh is

   function Check_End return Boolean;
   --  Called when an end sequence is required. In the absence of an error
   --  situation, Token contains Tok_End on entry, but in a missing end
   --  case, this may not be the case. Pop_End_Context is used to determine
   --  the appropriate action to be taken. The returned result is True if
   --  an End sequence was encountered and False if no End sequence was
```

```
      --  present. This occurs if the END keyword encountered was determined
      --  to be improper and deleted (i.e. Pop_End_Context set End_Action to
      --  Skip_And_Reject). Note that the END sequence includes a semicolon,
      --  except in the case of END RECORD, where a semicolon follows the END
      --  RECORD, but is not part of the record type definition itself.

      procedure End_Skip;
      --  Skip past an end sequence. On entry Token contains Tok_End, and we
      --  we know that the end sequence is syntactically incorrect, and that
      --  an appropriate error message has already been posted. The mission is
      --  simply to position the scan pointer to be the best guess of the
      --  position after the end sequence. We do not issue any additional
      --  error messages while carrying this out.

      procedure End_Statements;
      --  Called when an end is required or expected to terminate a sequence
      --  of statements. The caller has already made an appropriate entry in
      --  the Scope.Table to describe the expected form of the end. This can
      --  only be used in cases where the only appropriate terminator is end.

      procedure Pop_End_Context;
      --  Pop_End_Context is called after processing a construct, to pop
      --  the top entry off the end stack. It decides on the appropriate action
      --  to take, signalling the result by setting End_Action as described in
      --  the global variable section.

   end Endh;

   ------------------------------------
   -- Resynchronization After Errors --
   ------------------------------------

   --  These procedures are used to resynchronize after errors. Following an
   --  error which is not immediately locally recoverable, the exception
   --  Error_Resync is raised. The handler for Error_Resync typically calls
   --  one of these recovery procedures to resynchronize the source position
   --  to a point from which parsing can be restarted.

   --  Note: these procedures output an information message that tokens are
   --  being skipped, but this message is output only if the option for
   --  Multiple_Errors_Per_Line is set in Options.

   package Sync is

      procedure Resync_Choice;
      --  Used if an error occurs scanning a choice. The scan pointer is
      --  advanced to the next vertical bar, arrow, or semicolon, whichever
      --  comes first. We also quit if we encounter an end of file.

      procedure Resync_Expression;
      --  Used if an error is detected during the parsing of an expression.
      --  It skips past tokens until either a token which cannot be part of
      --  an expression is encountered (an expression terminator), or if a
      --  comma or right parenthesis or vertical bar is encountered at the
      --  current parenthesis level (a parenthesis level counter is maintained
      --  to carry out this test).

      procedure Resync_Past_Semicolon;
      --  Used if an error occurs while scanning a sequence of declarations.
      --  The scan pointer is positioned past the next semicolon and the scan
      --  resumes. The scan is also resumed on encountering a token which
      --  starts a declaration (but we make sure to skip at least one token
      --  in this case, to avoid getting stuck in a loop).
```

```
      procedure Resync_Past_Semicolon_Or_To_Loop_Or_Then;
      --  Used if an error occurs while scanning a sequence of statements.
      --  The scan pointer is positioned past the next semicolon, or to the
      --  next occurrence of either then or loop, and the scan resumes.

      procedure Resync_To_When;
      --  Used when an error occurs scanning an entry index specification.
      --  The scan pointer is positioned to the next WHEN (or to IS or
      --  semicolon if either of these appear before WHEN, indicating
      --  another error has occurred).

      procedure Resync_Semicolon_List;
      --  Used if an error occurs while scanning a parenthesized list of items
      --  separated by semicolons. The scan pointer is advanced to the next
      --  semicolon or right parenthesis at the outer parenthesis level, or
      --  to the next is or RETURN keyword occurence, whichever comes first.

      procedure Resync_Cunit;
      --  Synchronize to next token which could be the start of a compilation
      --  unit, or to the end of file token.

   end Sync;

   -----------------------------
   -- Token Scan Routines --
   -------------------------

   --  Routines to check for expected tokens

   package Tchk is

      --  Procedures with names of the form T_xxx, where Tok_xxx is a token
      --  name, check that the current token matches the required token, and
      --  if so, scan past it. If not, an error is issued indicating that
      --  the required token is not present (xxx expected). In most cases, the
      --  scan pointer is not moved in the not-found case, but there are some
      --  exceptions to this, see for example T_Id, where the scan pointer is
      --  moved across a literal appearing where an identifier is expected.

      procedure T_Abort;
      procedure T_Arrow;
      procedure T_At;
      procedure T_Body;
      procedure T_Box;
      procedure T_Colon;
      procedure T_Colon_Equal;
      procedure T_Comma;
      procedure T_Dot_Dot;
      procedure T_For;
      procedure T_Greater_Greater;
      procedure T_Identifier;
      procedure T_In;
      procedure T_Is;
      procedure T_Left_Paren;
      procedure T_Loop;
      procedure T_Mod;
      procedure T_New;
      procedure T_Of;
      procedure T_Or;
      procedure T_Private;
      procedure T_Range;
      procedure T_Record;
```

```
      procedure T_Right_Paren;
      procedure T_Semicolon;
      procedure T_Then;
      procedure T_Type;
      procedure T_Use;
      procedure T_When;
      procedure T_With;

      --  Procedures have names of the form TF_xxx, where Tok_xxx is a token
      --  name check that the current token matches the required token, and
      --  if so, scan past it. If not, an error message is issued indicating
      --  that the required token is not present (xxx expected).

      --  If the missing token is at the end of the line, then control returns
      --  immediately after posting the message. If there are remaining tokens
      --  on the current line, a search is conducted to see if the token
      --  appears later on the current line, as follows:

      --  A call to Scan_Save is issued and a forward search for the token
      --  is carried out. If the token is found on the current line before a
      --  semicolon, then it is scanned out and the scan continues from that
      --  point. If not the scan is restored to the point where it was missing.

      procedure TF_Arrow;
      procedure TF_Is;
      procedure TF_Loop;
      procedure TF_Return;
      procedure TF_Semicolon;
      procedure TF_Then;
      procedure TF_Use;

   end Tchk;

   ----------------------
   -- Utility Routines --
   ----------------------

   package Util is

      function Bad_Spelling_Of (T : Token_Type) return Boolean;
      --  This function is called in an error situation. It checks if the
      --  current token is an identifier whose name is a plausible bad
      --  spelling of the given keyword token, and if so, issues an error
      --  message, sets Token from T, and returns True. Otherwise Token is
      --  unchanged, and False is returned.

      procedure Check_Misspelling_Of (T : Token_Type);
      pragma Inline (Check_Misspelling_Of);
      --  This is similar to the function above, except that it does not
      --  return a result. It is typically used in a situation where any
      --  identifier is an error, and it makes sense to simply convert it
      --  to the given token if it is a plausible misspelling of it.

      procedure Check_95_Keyword (Token_95, Next : Token_Type);
      --  This routine checks if the token after the current one matches the
      --  Next argument. If so, the scan is backed up to the current token
      --  and Token_Type is changed to Token_95 after issuing an appropriate
      --  error message ("(Ada 83) keyword xx cannot be used"). If not,
      --  the scan is backed up with Token_Type unchanged. This routine
      --  is used to deal with an attempt to use a 95 keyword in Ada 83
      --  mode. The caller has typically checked that the current token,
      --  an identifier, matches one of the 95 keywords.
```

```
procedure Check_Simple_Expression (E : Node_Id);
--  Given an expression E, that has just been scanned, so that Expr_Form
--  is still set, outputs an error if E is a non-simple expression. E is
--  not modified by this call.

procedure Check_Simple_Expression_In_Ada_83 (E : Node_Id);
--  Like Check_Simple_Expression, except that the error message is only
--  given when operating in Ada 83 mode, and includes "in Ada 83".

function Check_Subtype_Mark (Mark : Node_Id) return Node_Id;
--  Called to check that a node representing a name (or call) is
--  suitable for a subtype mark, i.e, that it is an identifier or
--  a selected component. If so, or if it is already Error, then
--  it is returned unchanged. Otherwise an error message is issued
--  and Error is returned.

function Comma_Present return Boolean;
--  Used in comma delimited lists to determine if a comma is present, or
--  can reasonably be assumed to have been present (an error message is
--  generated in the latter case). If True is returned, the scan has been
--  positioned past the comma. If False is returned, the scan position
--  is unchanged. Note that all comma-delimited lists are terminated by
--  a right paren, so the only legitimate tokens when Comma_Present is
--  called are right paren and comma. If some other token is found, then
--  Comma_Present has the job of deciding whether it is better to pretend
--  a comma was present, post a message for a missing comma and return
--  True, or return False and let the caller diagnose the missing right
--  parenthesis.

procedure Discard_Junk_Node (N : Node_Id);
procedure Discard_Junk_List (L : List_Id);
pragma Inline (Discard_Junk_Node);
pragma Inline (Discard_Junk_List);
--  These procedures do nothing at all, their effect is simply to discard
--  the argument. A typical use is to skip by some junk that is not
--  expected in the current context.

procedure Ignore (T : Token_Type);
--  If current token matches T, then give an error message and skip
--  past it, otherwise the call has no effect at all. T may be any
--  reserved word token, or comma, left or right paren, or semicolon.

function Is_Reserved_Identifier return Boolean;
--  Test if current token is a reserved identifier. This test is based
--  on the token being a keyword and being spelled in typical identifier
--  style (i.e. starting with an upper case letter).

procedure No_Constraint;
--  Called in a place where no constraint is allowed, but one might
--  appear due to a common error (e.g. after the type mark in a procedure
--  parameter. If a constraint is present, an error message is posted,
--  and the constraint is scanned and discarded.

function No_Right_Paren (Expr : Node_Id) return Node_Id;
--  Function to check for no right paren at end of expression, returns
--  its argument if no right paren, else flags paren and returns Error.

procedure Push_Scope_Stack;
pragma Inline (Push_Scope_Stack);
--  Push a new entry onto the scope stack. Scope.Last (the stack pointer)
--  is incremented. The Junk field is preinitialized to False. The caller
--  is expected to fill in all remaining entries of the new new top stack
--  entry at Scope.Table (Scope.Last).
```

```
   procedure Pop_Scope_Stack;
   --  Pop an entry off the top of the scope stack. Scope_Last (the scope
   --  table stack pointer) is decremented by one. It is a fatal error to
   --  try to pop off the dummy entry at the bottom of the stack (i.e.
   --  Scope.Last must be non-zero at the time of call).

   function Separate_Present return Boolean;
   --  Determines if the current token is either Tok_Separate, or an
   --  identifier that is a possible misspelling of "separate" followed
   --  by a semicolon. True is returned if so, otherwise False.

   function Token_Is_At_Start_Of_Line return Boolean;
   pragma Inline (Token_Is_At_Start_Of_Line);
   --  Determines if the current token is the first token on the line

end Util;

---------------------------------------
-- Specialized Syntax Check Routines --
---------------------------------------

function Prag (Pragma_Node : Node_Id; Semi : Source_Ptr) return Node_Id;
--  This function is passed a tree for a pragma that has been scanned out.
--  The pragma is syntactically well formed according to the general syntax
--  for pragmas and the pragma identifier is for one of the recognized
--  pragmas. It performs specific syntactic checks for specific pragmas.
--  The result is the input node if it is OK, or Error otherwise. The
--  reason that this is separated out is to facilitate the addition
--  of implementation defined pragmas. The second parameter records the
--  location of the semicolon following the pragma (this is needed for
--  correct processing of the List and Page pragmas). The returned value
--  is a copy of Pragma_Node, or Error if an error is found.

--------------------------
-- Subsidiary Routines --
--------------------------

procedure Labl;
--  This procedure creates implicit label declarations for all label that
--  are declared in the current unit. Note that this could conceptually
--  be done at the point where the labels are declared, but it is tricky
--  to do it then, since the tree is not hooked up at the point where the
--  label is declared (e.g. a sequence of statements is not yet attached
--  to its containing scope at the point a label in the sequence is found)

procedure Load;
--  This procedure loads all subsidiary units that are required by this
--  unit, including with'ed units, specs for bodies, and parents for child
--  units. It does not load bodies for inlined procedures and generics,
--  since we don't know till semantic analysis is complete what is needed.

-----------
-- Stubs --
-----------

--  The package bodies can see all routines defined in all other subpackages

use Ch2;
use Ch3;
use Ch4;
use Ch5;
use Ch6;
```

```
   use Ch7;
   use Ch8;
   use Ch9;
   use Ch10;
   use Ch11;
   use Ch12;
   use Ch13;

   use Endh;
   use Tchk;
   use Sync;
   use Util;

   package body Ch2 is separate;
   package body Ch3 is separate;
   package body Ch4 is separate;
   package body Ch5 is separate;
   package body Ch6 is separate;
   package body Ch7 is separate;
   package body Ch8 is separate;
   package body Ch9 is separate;
   package body Ch10 is separate;
   package body Ch11 is separate;
   package body Ch12 is separate;
   package body Ch13 is separate;

   package body Endh is separate;
   package body Tchk is separate;
   package body Sync is separate;
   package body Util is separate;

   function Prag (Pragma_Node : Node_Id; Semi : Source_Ptr) return Node_Id
     is separate;

   procedure Labl is separate;
   procedure Load is separate;

   ---------
   -- Par --
   ---------

--  This function is the parse routine called at the outer level. It parses
--  the current compilation unit and adds implicit label declarations.

begin
   --  Deal with configuration pragmas case first

   if Configuration_Pragmas then
      declare
         Ecount  : constant Int := Errors_Detected;
         Pragmas : List_Id := Empty_List;
         P_Node  : Node_Id;

      begin
         loop
            if Token = Tok_EOF then
               return Pragmas;

            elsif Token /= Tok_Pragma then
               Error_Msg_SC ("only pragmas allowed in gnat.adc");
               return Error_List;

            else
```

```
            P_Node := P_Pragma;

            if Errors_Detected > Ecount then
               return Error_List;
            end if;

            if Chars (P_Node) > Last_Configuration_Pragma_Name
              and then Chars (P_Node) /= Name_Source_Reference
            then
               Error_Msg_SC
                 ("only configuration pragmas allowed in gnat.adc");
               return Error_List;
            end if;

            Append (P_Node, Pragmas);
         end if;
      end loop;
   end;

   --  Normal case of compilation unit

   else
      Save_Ada_83_Mode := Check_Ada_95 (File_Name (Current_Source_File));

      --  Special processing for language defined units. For this purpose
      --  we do NOT consider the renamings in annex J as predefined. That
      --  allows users to compile their own versions of these files, and
      --  in particular, in the VMS implementation, the DEC versions can
      --  be substituted for the standard Ada 95 versions.

      if Is_Predefined_File_Name
           (Fname => File_Name (Current_Source_File),
            Renamings_Included => False)
      then
         --  If this is the main unit, disallow compilation unless the -gnatg
         --  (GNAT mode) switch is set (from a user point of view, the rule is
         --  that language defined units cannot be recompiled).

         --  However, an exception is s-rpc, and its children. We test this
         --  by looking at the character after the minus, the rule is that
         --  System.RPC and its children are the only children in System
         --  whose second level name can start with the letter r.

         Get_Name_String (File_Name (Current_Source_File));

         if (Name_Len < 3 or else Name_Buffer (1 .. 3) /= "s-r")
           and then Current_Source_Unit = Main_Unit
           and then not GNAT_Mode
           and then Operating_Mode = Generate_Code
         then
            Error_Msg_SC ("language defined units may not be recompiled");
         end if;
      end if;

      --  Initialize scope table and other parser control variables

      Compiler_State := Parsing;
      Scope.Init;
      Scope.Increment_Last;
      Scope.Table (0).Etyp := E_Dummy;
      SIS_Entry_Active := False;
      Last_Resync_Point := No_Location;
```

```
      Label_List := New_Elmt_List;
      Unit_Node := P_Compilation_Unit;

      --  Now that we have completely parsed the source file, we can
      --  complete the source file table entry.

      Complete_Source_File_Entry;

      --  An internal error check, the scope stack should now be empty

      pragma Assert (Scope.Last = 0);

      --  Remaining steps are to create implicit label declarations and to
      --  load required subsidiary sources. These steps are required only
      --  if we are doing semantic checking.

      if Operating_Mode /= Check_Syntax or else Debug_Flag_F then
         Par.Labl;
         Par.Load;
      end if;

      --  Restore settings of switches saved on entry

      Ada_83 := Save_Ada_83_Mode;
      Ada_95 := not Ada_83;
      Set_Comes_From_Source_Default (False);
      return Empty_List;
   end if;

end Par;


-----------------------
-- REVISION HISTORY --
-----------------------

--  ----------------------------
--  revision 1.114
--  date: Mon Apr 27 08:17:00 1998;  author: dewar
--  Remove unused withs
--  ----------------------------
--  revision 1.115
--  date: Sun Jun 21 11:37:37 1998;  author: dewar
--  Minor reformatting
--  ----------------------------
--  revision 1.116
--  date: Mon Aug 10 17:36:31 1998;  author: dewar
--  Remove use of Features
--  (Is_Bad_Spelling): Moved to g-speche.ads
--  ----------------------------
--  New changes after this line.  Each line starts with: "--  "
```